Table 1: Runtime analysis for base case

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | | **Total Cost** |
| FOR each line, csvLine, in csv file | 2 | N | | 2N |
| CREATE new temporary course item, curCourse | 1 | N | | N |
| SET variable, lineSize, to number of items found in line | 1 | N | | N |
| SET curCourse ID as item 0 in csvLine | 1 | N | | N |
| IF lineSize < 2 | 1 | N | | N |
| SET curCourse name as item 1 in csvLine | 1 | N | | N |
| IF size >2 | 1 | N | | N |
| FOR items at csvLine positions 2 to lineSize | 2 |  | | N2-1 |
| CREATE Course, tempCourse with ID equal to the current prerequisite courseID | 1 |  | |  |
| PUSHBACK to add tempCourse to curCourse prerequisite vector | 1 |  | |  |
| CALL ADT-specific function, insert curCourse to courseList (use pushback for vector) | X | N | | N\*X |
| Total Cost | | |  | |
| Runtime | | | O(N2) | |
|  | | |  | |

Table 2: Runtime analysis for Hash Table Insert Function

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | **Total Cost** |
| SET key equal to the hash value of addCourse’s course ID | 1 | 1 | 1 |
| CREATE node object, oldNode | 1 | 1 | 1 |
| SET oldNode equal to the node found at key | 1 | 1 | 1 |
| IF oldNode is null | 1 | 1 | 1 |
| ELSE IF the old node is not used anymore | 1 | 1 | 1 |
| WHILE the node after oldNode is not null | 1 | N+1 | N+1 |
| SET old node to its subsequent node | 1 | N | N |
| SET the node after oldNode to a new node with addCourse and the calculated key | 1 | 1 | 1 |
| Total Cost | | | 2N+7 |

Table 3: Runtime analysis for binary search tree Insert function

|  |  |  |  |
| --- | --- | --- | --- |
| **Code** | **Line Cost** | **# Executions** | **Total Cost** |
| IF root is null | 1 | 1 | 1 |
| IF insertNode’s course’s courseId is greater than addCourse’s courseId | 1 | 1 | 1 |
| IF the node to the left of insertNode is null | 1 | 1 | 1 |
| RECURSE addNode with insertNode equal to its left node, and addCourse | 1 | N/2 | N/2 |
| IF insertNode’s course’s courseId is less than or equal to than course’s courseId | 1 | 1 | 1 |
| IF insertNode’s course’s courseId is less than or equal to than course’s courseId | 1 | 1 | 1 |
| IF the node to the right of insertNode is null | 1 | 1 | 1 |
| RECURSE addNode with insertNode equal to its right node, and the bid argument | 1 | N/2 | N/2 |
| Total Cost | | | N+6 |
| Runtime | | | O(N) |

Table 4: Big-O notation for worst case growth rate analyses performed for vectors, hash tables, and binary search trees

|  |  |  |
| --- | --- | --- |
|  | **Function** | **Big-O value** |
| **Vector** |  | O(N2) |
| **HashTable** |  | O(N2) |
| **Binary Search Tree** |  | O(N2) |

Vectors, in comparison with arrays, are useful in cases like this where the size of the vector is initially unknown. This makes it quite easy to add and remove items to the vector without wasting space. They are also quite straightforward to iterate over since each item is stored sequentially. A disadvantage of vectors (in contrast with binary search trees) is they are not inherently sorted. To access the vector values in alphanumeric order, as is the case in this project, a sorting algorithm must be performed. In the case of printCourseInformation, I chose to use Quicksort to sort the vector of courses to avoid linear searching, which can very inefficient in the worst-case scenario. Quicksort itself has a quadratic worst-case growth rate (Vahid F. , Lysecky, Wheatland, & Siu, 2019), which is ultimately why the vector has the largest Big-O value for printCourseInformation.

A major advantage hash tables generally have is the search speed based on the way items are mapped. When collisions do not occur, hash tables can have a constant growth rate for insertion, deletion, and searching for items (Vahid F. , Lysecky, Wheatland, & Siu, 2019). Even when collisions do occur, the table can use probing or chaining to complete these operations relatively quickly (although as a table fills up and collisions become more frequent, the complexity of the program will increase significantly). A disadvantage of hash tables, as previously mentioned, is the inability to sort them. Since they are mapped based on a hash value, their order will not be alphanumeric. Since 1 out of the 3 key functions relies on the items to be printed alphanumerically, this should discount hash tables as a contender for the project.

Although the worst-case-scenario does not reflect this, binary search trees have a complexity advantage over vectors as they approach being perfect. In a perfect binary search tree with 100 courses, the maximum number of searches required to find a course would be 7 (logN) (Vahid F. , Lysecky, Wheatland, & Siu, 2019). Binary search trees are inherently structured in a way that makes accessing items in alphanumeric order very straightforward and efficient. As previously mentioned, this is a huge advantage they have over hash tables and vectors (and why, ultimately, hash tables were converted into binary search trees to print the sample schedule). A downside of binary search trees is their efficiency dependency on the node selected for the root. As the root gets further away from the median value of the list, the tree will get more and more imperfect.

I am still getting used to understanding how to calculate runtime complexities, so I am not entirely confident in my conclusion based on Big-O value alone that binary search trees are the most optimal ADT to work with. However, looking at other advantages that binary search trees have over hash tables and vectors in this context reaffirm that they are the right choice for this project.

Bibliography

Vahid, F., Lysecky, S., Wheatland, N., & Siu, R. (2019). *CS 300: Data Structures and Algorithms.* Zyante Inc.